

# EFANNA : An Extremely Fast Approximate Nearest Neighbor Search Algorithm Based on kNN Graph

Cong Fu, Deng Cai

**Abstract**—Approximate nearest neighbor (ANN) search is a fundamental problem in many areas of data mining, machine learning and computer vision. The performance of traditional hierarchical structure (tree) based methods decreases as the dimensionality of data grows, while hashing based methods usually lack efficiency in practice. Recently, the graph based methods have drawn considerable attention. The main idea is that a *neighbor of a neighbor is also likely to be a neighbor*, which we refer as *NN-expansion*. These methods construct a  $k$ -nearest neighbor ( $k$ NN) graph offline. And at online search stage, these methods find candidate neighbors of a query point in some way (e.g., random selection), and then check the neighbors of these candidate neighbors for closer ones iteratively. Despite some promising results, there are mainly two problems with these approaches: 1) These approaches tend to converge to local optima. 2) Constructing a  $k$ NN graph is time consuming. We find that these two problems can be nicely solved when we provide a good initialization for NN-expansion. In this paper, we propose EFANNA, an extremely fast approximate nearest neighbor search algorithm based on  $k$ NN Graph. Efanna nicely combines the advantages of hierarchical structure based methods and nearest-neighbor-graph based methods. Extensive experiments have shown that EFANNA outperforms the state-of-art algorithms both on approximate nearest neighbor search and approximate nearest neighbor graph construction. To the best of our knowledge, EFANNA is the fastest algorithm so far both on approximate nearest neighbor graph construction and approximate nearest neighbor search. A library EFANNA based on this research is released on Github.

**Index Terms**—Approximate nearest neighbor search, approximate kNN graph construction.

## 1 INTRODUCTION

Nearest neighbor search plays an important role in many applications of data mining, machine learning and computer vision. When dealing with sparse data (e.g., document retrieval), one can use advanced index structures (e.g., inverted index) to solve this problem. However, for data with dense features, the cost for finding the exact nearest neighbor is  $O(N)$ , where  $N$  is the number of points in the database. It's very time consuming when the data set is large. So people turn to Approximate Nearest neighbor (ANN) search in practice [1], [5]. Many work has been done to carry out the ANN search with high accuracy but low computational complexity.

There are mainly two types of methods in ANN search. The first type methods are hierarchical structure (tree) based methods, such as KD-tree [3], [13], Randomized KD-tree [29], K-means tree [14]. These methods perform very well when the dimension of the data is relatively low. However, the performance decreases dramatically as the dimension of the data increases [29]. The second type methods are hashing based methods, such as Locality Sensitive Hashing (LSH) [16], Spectral Hashing [33], Iterative Quantization [35] and so on.

Please see [32] for a detailed survey on various hashing methods. These methods generate binary codes for high dimensional real vectors while try to preserve the similarity among original real vectors. Thus, all the real vectors fall into different hashing buckets (with different binary codes). Ideally, if neighbor vectors fall into the same bucket or the nearby buckets (measured by the hamming distance of two binary codes), the hashing based methods can efficiently retrieve the nearest neighbors of a query point. However, there is no guarantee that all the neighbor vectors will fall into the nearby buckets. To ensure the high *recall* (the number of true neighbors within the returned points set divides by the number of required neighbors), one needs to examine many hashing buckets (i.e., enlarge the search radius in hamming space), which results a high computational complexity. Please see [21] for a detailed analysis.

Recently, graph based methods have drawn considerable attention [11], [17], [21]. The essential idea behind these approaches is that a *neighbor of a neighbor is also likely to be a neighbor*, which we refer as *NN-expansion*. These methods construct a  $k$ -nearest neighbor ( $k$ NN) graph offline. And at online search stage, these methods find the candidate neighbors of a query point in some way (e.g., random selection [11], [17]), and then check the neighbors of these candidate neighbors for closer ones iteratively. One problem of this approach is that the NN-expansion is easily to converge to local optima and

C. Fu and D. Cai are with the State Key Lab of CAD&CG, College of Computer Science, Zhejiang University, Hangzhou, Zhejiang, China, 310058. Email: 15267003518@163.com, dengcai@cad.zju.edu.cn.

result in a low recall. [21] tries to solve this problem by providing better initialization for a query point. Instead of random selection, [21] uses hashing based methods for initialization. This approach is named as Iterative Expanding Hashing (IEH) and achieves significant better results than the corresponding hashing based methods.

Another challenge on using graph based methods is the high computational cost in building the  $k$ NN graph, especially when the database is large. There are many efforts that have been put into reducing the time complexity of  $k$ NN graph construction. [4], [9], [31], [27] try to speed up an exact  $k$ NN graph construction. However, these approaches are still not efficient enough in the context of big data. Instead of building an exact  $k$ NN graph, recent researchers try to build an approximated  $k$ NN graph efficiently. The *NN-expansion* idea again can be used to build an approximated  $k$ NN graph. [12] proposed NN-descent to efficiently build an approximate  $k$ NN graph. [8], [15], [36], [30] try to build an approximate  $k$ NN graph in a divide-and-conquer manner. Their algorithms mainly contain three phrases. Firstly, they divide the whole data set into small subsets multiple times. Secondly, they do brute force search within the subsets and get lots of overlapping subgraphs. Finally, they merge the subgraphs and refine the graph with techniques similar to NN-expansion. Although an approximated  $k$ NN graph can be efficiently constructed, there are no formal study on how the performance of graph based search methods will be affected if one uses an approximated  $k$ NN graph instead of an exact  $k$ NN graph.

To tackle above problems, we propose a novel graph-based approximate nearest neighbor search framework EFANNA in this paper. EFANNA is an abbreviation for two algorithms: Extremely Fast Approximate  $k$ -Nearest Neighbor graph construction Algorithm and Extremely Fast Approximate Nearest Neighbor search Algorithm based on  $k$ NN graph. Our algorithm is based on a simple observation: the performance of both NN-expansion and NN-descent is very sensitive with the initialization.

Our EFANNA index contains two parts: the multiple randomized hierarchical structures (e.g., randomized truncated KD-tree) and an approximate  $k$ -nearest neighbor graph.

At offline stage, EFANNA divides the data set multiple times into a number of subsets in a fast and hierarchical way, producing multiple randomized hierarchical structures. Then EFANNA constructs an approximate  $k$ NN graph by conquering bottom-up along the structures. When conquering, EFANNA takes advantage of the structures to locate the closest possible neighbors, and use these candidates to update the graph, which reduces the computation cost than using all the points in the subtree. Finally we refine the graph similar to NN-descent [12], which is based on the NN-expansion idea and optimized with techniques like local join, sampling, and early termination [12].

At online search stage, EFANNA first search in the

hierarchical structures to get candidate neighbors for a given query. Then EFANNA refines the results using NN-expansion on the approximate  $k$ NN graph. Extensive experimental results show that our approach outperforms the the-state-of-the-art approximate nearest neighbor search algorithms significantly.

It is worthwhile to highlight the contributions of our paper as follows:

- EFANNA outperforms state-of-the-art ANN search algorithms significantly. Particularly, EFANNA outperforms Flann [25], one of the most popular ANN search library, in index size, search speed and search accuracy.
- EFANNA can build an approximate  $k$ NN graph with hundreds times speed-up over brute-force graph building on million scale datasets. Considering many unsupervised and semi-supervised machine learning algorithms [2], [28] are based on a nearest neighbor graph. EFANNA provides the possibility to examine the effectiveness of all these algorithms on large scale datasets.
- We show by experimental results that with an approximate  $k$ NN graph of low accuracy constructed by EFANNA, graph-based ANN search methods (e.g., EFANNA) still perform very good. This is because the "error" neighbors of approximate  $k$ NN graph constructed by EFANNA are actually neighbors a little farther. This property is never explored in the previous work.

The remainder of this paper is organized as follows. In Section 2 we will introduce some related work. Our EFANNA algorithm is presented in section 3. In section 4, we will report the experimental results and show the performance of EFANNA comprehensively. In section 5 we will talk about our open library and in section 6 we will draw a conclusion.

## 2 RELATED WORK

Nearest neighbor search [34] has been a hot topic during the last decades. Due to the intrinsic difficulty of exact nearest neighbor search, the approximate nearest neighbor (ANN) search algorithms [1], [19] are widely studied and the researchers expect to sacrifice a little searching accuracy to lower the time cost as much as possible.

Hierarchical index based (tree based) algorithms, such as KD-tree [13], have gained early success on approximate nearest neighbor search problems. However, it's proved to be inefficient when the dimensionality of data grows high. Many new hierarchical structure based methods [29], [7], [26] are presented to address this limitation. Randomized KD-tree [29] and Kmeans tree [26] are absorbed into a well-known open source library FLANN [24], which has gained wide popularity.

Hashing based algorithms [16], [33] aim at finding proper ways to generate binary codes for data points and preserve their similarity in original feature space. These methods can be treated as dividing the data

space with multiple hyperplanes and representing each resulting polyhedron with a binary code. Learning the hashing functions with different constraints will result in different partition of data space. One of the most famous algorithms is Locality Sensitive Hashing (LSH) [16], which is essentially based on random projection [6]. Many other variants [33], [35], [18], [20] are proposed based on different constraints. And the constraints reflect what they think is the proper way to partition the data space.

Both the hashing based methods and tree based methods have the same goal. They expect to put neighbors into the same hashing bucket (or node). However, there is no theoretical guarantee of this expectation. To increase the search *recall* (the number of true neighbors within the returned points divides by the number of required neighbors), one needs to check the “nearby” buckets or nodes. With high dimensional data, one polyhedron may have a large amount of neighbor polyhedrons, (for example, a bucket with 32 bit hashing code has 32 neighbor buckets with 1 hamming radius distance), which makes locating true neighbors hard [21].

Recently graph based techniques have drawn considerable attention [11], [17], [21]. The main idea of these methods is *a neighbor of a neighbor is also likely to be a neighbor*, which we refer as *NN-expansion*. At offline stage, they need to build a  $k$ NN graph, which can be regraded as a big table recording the top  $k$  closest neighbors of each point in database. At online stage, given a query point, they first assign the query some points as initial candidate neighbors, and then check the neighbors of the neighbors iteratively to locate closer neighbors. Graph Nearest neighbor Search (GNNS) [17] randomly generate the initial candidate neighbors while Iterative Expanding Hashing (IEH) [21] uses hashing algorithms to generate the initial candidate neighbors.

Since all the graph based methods need a  $k$ NN graph as the index structure, how to build a  $k$ NN graph efficiently became a crucial problem, especially when the database is large. Many work has been done on building either exact or approximate  $k$ NN graph. [4], [9], [27], [31] try to build an exact  $k$ NN graph quickly. However, these approaches are still not efficient enough in the context of big data. Recently, researchers try to build an approximated  $k$ NN graph efficiently. Again, the NN-expansion idea can be used here. [12] proposed an *NN-descent* algorithm to efficiently build an approximate  $k$ NN graph. The basic idea of NN-descent is similar to NN-expansion but the details are different. NN-descent uses many techniques (e.g., **Local join** and **Sampling**) to efficiently refine the graph. Please see [12] for details.

Instead of initializing the  $k$ NN graph randomly, [8], [15], [36], [30] uses some divide-and-conquer methods. Their initialization contains two parts. Firstly, they divide the whole data set into small subsets multiple times. Secondly, they do brute force search within the subsets and get lots of overlapping subgraphs. These subgraphs can be merged together to serve as the initialization

---

#### Algorithm 1 EFANNA Search Algorithm

---

**Input:** data set  $D$ , query vector  $q$ , the number  $K$  of required nearest neighbors, EFANNA index (including tree set  $S_{tree}$  and  $k$ NN graph  $G$ ), search-to depth  $S_{depth}$ , the candidate pool size  $P$ , the iteration number  $I$ .

**Output:** approximate nearest neighbor set  $ANNS$  of the query

- 1:  $iter = 0$
- 2: candidate set  $C = \emptyset$
- 3: **for all** tree  $i$  in  $S_{tree}$  **do**
- 4:   search for candidate neighbors of  $q$  in tree  $i$  to depth  $S_{depth}$ , add all the points in the subtree into  $C$ .
- 5: **end for**
- 6: keep  $P$  points in  $C$  which are closest to  $q$ .
- 7: **while**  $iter < I$  **do**
- 8:   candidate set  $CC = \emptyset$
- 9:   **for all** point  $n$  in  $C$  **do**
- 10:      $S_{neigh}$  is the neighbors of point  $n$  based on  $G$ .
- 11:     **for all** point  $nn$  in  $S_{neigh}$  **do**
- 12:       put  $nn$  into  $CC$ .
- 13:     **end for**
- 14:   **end for**
- 15:   move all the points in  $CC$  to  $C$  and keep  $P$  points in  $C$  which are closest to  $q$ .
- 16:    $iter = iter + 1$
- 17: **end while**
- 18: return  $ANNS$  as the closet  $K$  points to  $q$  in  $C$ .

---

of  $k$ NN graph. The NN-expansion like techniques can then be used to refine the graph. The division step of [8] is based on a spectral bisection and they proposed two different versions, overlap and glue division. [36] use Anchor Graph Hashing [22] to produce the division. [15] uses recursive random division, dividing orthogonal to the principle direction of randomly sampled data in subsets. [30] uses random projection trees to partition the datasets.

[36] and [30] claim to outperform NN-descent [12] significantly. However, based on their reported results and our analysis, there seems a misunderstanding of NN-descent [12]. Actually, NN-descent is quite different than NN-expansion. The method compared in [36] and [30] should be NN-expansion instead of NN-descent. Please see Section 3.3.3 for details.

### 3 EFANNA ALGORITHMS FOR ANN SEARCH

We will introduce our EFANNA algorithms in this section. EFANNA algorithms include offline index building part and online ANN search algorithm. The EFANNA index contains two parts: multiple hierarchical structures (e.g., randomized truncated KD-tree) and an approximate  $k$ NN graph. We will first show how to use EFANNA index to carry out online ANN search. Then

**Algorithm 2** EFANNA Tree Building Algorithm

**Input:** the data set  $D$ , the number of trees  $T$ , the number of points in a leaf node  $K$ .

**Output:** the randomized truncated KD-tree set  $S$

---

```

1:
2: function BUILDTREE(Node, PointSet)
3:   if size of PointSet <  $K$  then
4:     return
5:   else
6:     Randomly choose dimension  $d$ .
7:     Calculate the median  $mid$  over PointSet on
       dimension  $d$ .
8:     Divide PointSet evenly into two subsets,
       LeftHalf and RightHalf, according to  $mid$ .
9:     BUILDTREE(Node.LeftChild, LeftHalf)
10:    BUILDTREE(Node.RightChild, RightHalf)
11:   end if
12:   return
13: end function
14:
15: for all  $i = 1$  to  $T$  do
16:   BUILDTREE(Rooti,  $D$ )
17:   Add Rooti to  $S$ .
18: end for

```

---

we will show how to build the EFANNA index in a divide-conquer-refinement manner.

### 3.1 ANN search with EFANNA index

EFANNA is a graph based method. The main idea is providing better initialization for NN-expansion to improve the performance significantly. The multiple hierarchical structures is used for initialization and the approximate  $k$ NN graph is used for NN-expansion.

There are many possible hierarchical structures (e.g. hierarchical clustering [25] or randomized division tree [10]) can be used in our index structure. In this paper, we only report the results using randomized *truncated* KD-tree. The details on the difference between this structure and the traditional randomized KD-tree [29] will be discussed in the next subsection. Based on this randomized truncated KD-tree, we can get the initial neighbor candidates given a query  $q$ . We then refine the result with NN-expansion, i.e., we check the neighbors of  $q$ 's neighbors according to the approximate  $k$ NN graph to get closer neighbors. Algorithm 1 shows the detailed procedure.

There are three essential parameters in our ANN search algorithm: search-to depth  $S_{depth}$ , the candidate pool size  $P$  and the iteration number  $I$ . We define the depth of tree root is 0, and increases downwards. In our experiment, we found  $I = 4$  is enough and thus we fixed  $I = 4$ . The trade-off between search speed and accuracy can be made through tuning parameters  $S_{depth}$  and  $P$ . A larger  $S_{depth}$  indicates smaller number of candidate points after line 5 in Algorithm 1, which makes line 6

**Algorithm 3** Hierarchical Divide-and-Conquer Algorithm ( $k$ NN Graph Initialization)

**Input:** the data set  $D$ , the  $k$  in approximate  $k$ NN graph, the randomized truncated KD-tree set  $S$  built with Algorithm 2, the conquer-to depth  $Dep$ .

**Output:** approximate  $k$ NN graph  $G$ .

---

```

1: %%Division step
2: Using Algorithm 2 to build tree, which leads to the
   input  $S$ 
3:
4: %% Conquer step
5:  $G = \emptyset$ 
6: for all point  $i$  in  $D$  do
7:   Candidate pool  $C = \emptyset$ 
8:   for all binary tree  $t$  in  $S$  do
9:     search in tree  $t$  with point  $i$  to the leaf node.
10:    add all the point in the leaf node to  $C$ .
11:     $d = \text{depth of the leaf node}$ 
12:    while  $d > Dep$  do
13:       $d = d - 1$ 
14:      Depth-first-search in the tree  $t$  with point
        $i$  to depth  $d$ . Suppose  $N$  is the non-leaf node on the
       search path with depth  $d$ . Suppose  $Sib$  is the child
       node of  $N$ . And  $Sib$  is not on the search path of point
        $i$ .
15:      Depth-first-search to the leaf node in the
       subtree of  $Sib$  with point  $i$ . Add all the points in
       the leaf node to  $C$ .
16:    end while
17:   end for
18:   Reserve  $K$  closest points to  $i$  in  $C$ .
19:
20:   Add  $C$  to  $G$ .
21: end for

```

---

faster and the whole algorithm faster. However, larger  $S_{depth}$  sacrifices the accuracy. On the other hand, a larger  $P$  sacrifices the search speed for high accuracy.

### 3.2 EFANNA Index Building Algorithms I : Tree Buidling

One part of the EFANNA index is the multiple hierarchical structures. There are many possible hierarchical structures (e.g. hierarchical clustering [25] or randomized division tree [10]) can be used. In this paper, we only report the results using randomized *truncated* KD-tree. Please see Algorithm 2 for details on building randomized truncated KD-trees.

The only difference between randomized *truncated* KD-tree and the traditional randomized KD-tree is that leaf node in our trees has  $K$  ( $K = 10$  in our experiments) points instead of 1. This change makes the tree building in EFANNA much faster than the the traditional randomized KD-tree. Please see Table 4 in the experiments for details.

The randomized truncated KD-tree built in this step

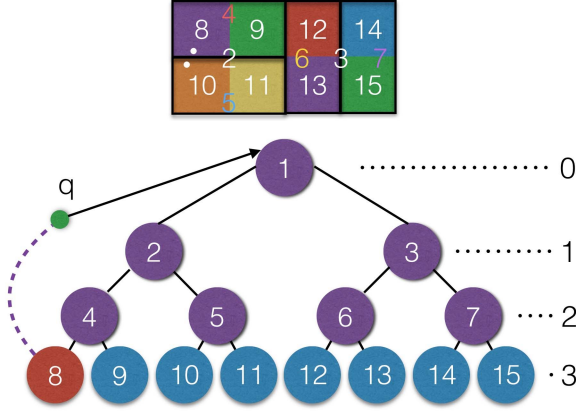


Fig. 1. An example for our hierarchical divide-and-conquer algorithm.

is used not only in the on-line search stage, but also in the approximate  $k$ NN graph construction stage. See the next section for details.

### 3.3 EFANNA Index Building Algorithms II : Approximate $k$ NN Graph Construction

The methodology of the  $k$ NN graph construction is similar with search. It contains two stage. At first stage, we regard the trees built in the previous part as multiple overlapping divisions over the data set, and we perform the conquering step along the tree structures to get an initial  $k$ NN graph. At second stage, we use NN-descent [12] to refine the  $k$ NN graph.

#### 3.3.1 Hierarchical Randomized Divide-and-Conquer

As we mentioned above, we try to provide a better initialization for NN-descent. A good initialization should produce an initial  $k$ NN graph with certain accuracy within short time. Divide-and-conquer is a straightforward strategy to achieve this goal. Traditional divide-and-conquer first breaks the problem into subproblems recursively until the subproblem is small and easy enough to solve. Then the solutions of subproblems are combined to get a solution to the original problem. As for approximate  $k$ NN graph construction, for example, in Fig 1, suppose that the whole data set is divided as the way the tree was constructed (Section 3.2). When conquering at level 3, a brute-force graph construction is carried out within each subset and a  $k$ NN graph with partial connectivity will be produced. Then at level 2, subset 8 and 9 will be merged to 4, and so will subset 10 and 11, 12 and 13, and 14 and 15. And when 8 and 9 are merged, distance between points in subset 8 and 9 should be calculated, and the graph will be updated respectively. So it is with other “merging”. Next, subset 4 and 5, 6 and 7 will be merged. This process is carried out recursively until we merge subset 2 and 3 to get a graph of full connectivity on the whole data set.

However, the computation complexity grows exponentially while the size of subsets to be conquered

#### Algorithm 4 Approximate $k$ NN Graph Refinement

**Input:** an initial approximate  $k$ -nearest neighbor graph  $G_{init}$ , data set  $D$ , maximum iteration number  $I$ , Candidate pool size  $P$ , new neighbor checking num  $L$ .

**Output:** an approximate  $k$ NN graph  $G$ .

```

1:  $iter = 0, G = G_{init}$ 
2: Graph  $G_{new}$  records all the new added candidate neighbors of each point.  $G_{new} = G_{init}$ .
3: Graph  $G_{old}$  records all the old candidate neighbors of each point at previous iterations.  $G_{old} = \emptyset$ 
4: Graph  $G_{rnew}$  records all the new added reverse candidate neighbors of each point.
5: Graph  $G_{roid}$  records all the old reverse candidate neighbors of each point.
6: while  $iter < I_{max}$  do
7:    $G_{rnew} = \emptyset, G_{roid} = \emptyset$ .
8:   for all point  $i$  in  $D$  do
9:      $NN_{new}$  is the neighbor set of point  $i$  in  $G_{new}$ .
10:     $NN_{old}$  is the neighbor set of point  $i$  in  $G_{old}$ .
11:    for all point  $j$  in  $NN_{new}$  do
12:      for all point  $k$  in  $NN_{new}$  do
13:        if  $j \neq k$  then
14:          compute the distance between  $j$  and
15:           $k$ .
16:          add  $k$  to  $j$ 's entry in  $G$ . mark  $k$  as
17:           $new$ .
18:          add  $j$  to  $k$ 's entry in  $G$  and  $G_{rnew}$ .
19:          mark  $j$  as  $new$ .
20:        end if
21:      end for
22:    for all point  $l$  in  $NN_{old}$  do
23:      compute the distance between  $j$  and  $l$ .
24:      add  $l$  to  $j$ 's entry in  $G$ . mark  $l$  as  $old$ .
25:      add  $j$  to  $l$ 's entry in  $G$  and  $G_{roid}$ .
26:      mark  $j$  as  $old$ .
27:    end for
28:  end for
29:  for all point  $i$  in  $D$  do
30:    Reserve the closest  $P$  points to  $i$  in respective
31:    entry of  $G$ .
32:  end for
33:   $G_{new} = G_{old} = \emptyset$ 
34:  for all point  $i$  in  $D$  do
35:     $l = 0$ .  $NN$  is the neighbor set of  $i$  in  $G$ .
36:    while  $l < L$  and  $l < P$  do
37:       $j = NN[l]$ .
38:      if  $j$  is marked as  $new$  then
39:        add  $j$  to  $i$ 's entry in  $G_{new}$ .
40:         $l = l + 1$ .
41:      else
42:        add  $j$  to  $i$ 's entry in  $G_{old}$ .
43:      end if
44:    end while
45:  end for
46:   $G_{new} = G_{new} \cup G_{rnew}$ .
47:   $G_{old} = G_{old} \cup G_{roid}$ 
48:   $iter = iter + 1$ 
49: end while

```

grows. To reduce computation complexity, we reduce the number of points involved in conquering at each level. For example, in Fig 1, if we know that point  $q$  in node (or subset) 8 is closer to the area of node 10, then we just need to consider the points in node 10 when conquering 4 and 5 at level 1 with  $q$ .

Now we regard the tree as a **multi-class classifier**, each leaf node can be treated as a class. This classifier may divide the data space like the rectangle in Fig. 1 does. Sometimes nearest neighbors (*e.g.* white points in area 8 and area 10) are close to each other but fit in different area according to this classifier with a discriminative plane (*e.g.* node 2) separating them. When conquering to some level (to certain non-leaf node), for each point in one subtree, we may just consider the “closest” possible leaf node in the sibling subtree. Because the rest leaf nodes are farther, the points in them are also likely to be farther, thus excluded from distance calculating.

In Fig .1, point  $q$  will be assigned label 8 when  $q$  is input into the tree classifier (use depth first search). When conquering at level 1, we want to know in subtree 5 which area of node 10 and 11 is closer to  $q$ . If area 10 is closer to  $q$ , the points in node 10 is very likely to be closer to  $q$  than 11. Now that the whole tree can be treated as a **multi-class classifier**, any subtree of it can be a **multi-class classifier**, too. To know which area is closer, we simply let the classifier of subtree 5 make the choice. By inputting  $q$  to the classifier, we will obtain a label between 10 and 11 for  $q$ . Suppose  $q$  is the white point in area 8, from the rectangle in Fig. 1, it’s clear that  $q$  will be labeled as 10. Therefore for  $q$ , when conquering at level 1, only points in node 10 will be involved.

In this way, for each point at each level, only the points in one closest leaf node will be considered, which reduces the computation complexity greatly and reserves accuracy. However, the graph we get from one divide-and-conquer process has full connectivity only if we conquer to root of the tree. Multiple randomized divide-and-conquer without conquering to root is also able to achieve full connectivity. And we find in our experiments that multiple randomized divide-and-conquer achieves better performance on initialization and the resulting tree structures also provide better initialization to ANN search as well.

Again there is a trade-off between accuracy of initial graph and time cost in parameter tuning. When conquer-to depth  $Dep$  is small (*i.e.* conquering to a level close to root), or when tree number  $T_c$  is larger, the accuracy is higher but time cost is higher. In our experiments, we use the randomized KD-tree as the hierarchical divide-and-conquer structure. See Algorithm 3 for details on randomized KD-tree divide-and-conquer algorithm).

### 3.3.2 Graph Refinement

We use the NN-descent proposed by [12] to refine the resulting graph we get from the divide-and-conquer step. The main idea is also to find better neighbors iteratively, however, different from NN-expansion, they proposed

several techniques to get much better performance. We rewrite their algorithms to make it easy to understand. See Algorithm 4 for details.

The pool size  $P$  and neighbor checking num  $L$  are essential parameters of this algorithm. Usually, Larger  $L$  and  $P$  will result in better accuracy but higher computation cost.

### 3.3.3 NN-expansion VS. NN-descent

Some approximate  $k$ NN graph construction methods [36] [30] claim to outperform NN-descent [12] significantly. However, based on their reported results and our analysis, there seems a misunderstanding of NN-descent [12]. Actually, NN-descent is quite different than NN-expansion. For given point  $p$ , NN-expansion assume the neighbors of  $p$ ’s neighbors are likely to be neighbors of  $p$ . While NN-descent thinks that  $p$ ’s neighbors are more likely to be neighbors of each other. Our experimental results have shown that NN-descent is much more efficient than NN-expansion in building approximate  $k$ NN graph. However, the NN-descent idea cannot be applied to ANN search.

## 3.4 Online index updating

EFANNA index building algorithm is easily to be extended to accept stream data. Firstly, when a new point arrived, we can insert it into the tree easily. And when the number of points in the inserted node exceeds given threshold, we just need to split the node. When the tree is unbalanced to some degree, we should adjust the tree structure, which is quite fast on large scale data. Secondly, the graph building algorithm can accept stream data as well, we can use the same algorithm we describe before. First we search in the tree for candidates, and use NN-descent to update the graph with the involved points. And this step is quite fast, too.

## 4 EXPERIMENTS

To demonstrate the effectiveness of the proposed method EFANNA, extensive experiments on large-scale data sets are reported in this section.

### 4.1 Data Set and Experiment Setting

The experiments were conducted on two popular real world data sets, SIFT1M and GIST1M<sup>1</sup>. The detailed information on the data sets is listed in TABLE 1. All the codes we used are written in C++ and compiled by g++4.9, and the only optimization option we allow is “O3” of g++. Parallelism and other optimization like SSE instruction are disabled. The experiment on SIFT1M is carried out on a machine with i7-3770K CPU and 16G memory, and GIST1M is on a machine with i7-4790K CPU and 32G memory.

1. Both two datasets can be downloaded at <http://corpus-texmex.irisa.fr/>

TABLE 1  
information on experiment data sets

data set	dimension	base number	query number
SIFT1M	128	1,000,000	10,000
GIST1M	960	1,000,000	1,000

## 4.2 Experiments on ANN Search

### 4.2.1 Evaluation Protocol

To measure the performance of ANN search of different algorithms, we used the well known *average recall* as the accuracy measurement [23]. Given a query point, all the algorithms are expected to return  $k$  points. Then we need to examine how many points in this returned set are among the true  $k$  nearest neighbors of the query. Suppose the returned set of  $k$  points given a query is  $R'$  and the true  $k$  nearest neighbors set of the query is  $R$ , the *recall* is defined as

$$recall(R') = \frac{|R' \cap R|}{|R'|}. \quad (1)$$

Then the *average recall* is averaging over all the queries. Since the sizes of  $R'$  and  $R$  are the same, the recall of  $R'$  is the same as the accuracy of  $R'$ .

We compare the performance of different algorithms by requiring different number of nearest neighbors of each query point, including 1-NN, 10-NN, 50-NN and 100-NN. In other words, the size of  $R$  (and  $R'$ ) will be 1, 10, 50 and 100 respectively.

### 4.2.2 Comparison Algorithms

To demonstrate the effectiveness of the proposed EFANNA approach, the following four state-of-the-art ANN search methods and brute-force method are compared in the experiment.

- 1) **brute-force**. We report the performance of brute-force search to show the advantages of using ANN search methods. To get different recall, we simply perform brute-force search on different percentage of the query number. For example, the brute-force search time on 90% queries of the origin query set stands for the brute-force search time of 90% average recall.
- 2) **flann**. FLANN is a well-known open source library for ANN search [25]. The Randomized KD-tree algorithm in FLANN provides state-of-the-art performance. In our experiments, we use 16 trees for SIFT1M and 32 trees for GIST1M. And we tune the “max-check” parameter to get the time-recall curve.
- 3) **GNNS**. GNNS is the first ANN search method using  $k$ NN graph [17]. Given a query, GNNS generates the initial candidates (neighbors) by random selection. Then GNNS uses the NN-expansion idea (*i.e.*, check the neighbors of the neighbors iteratively to locate closer neighbors) to refine the

result. The main parameters of GNNS are the size of the initial result and the iteration number. We fix the iteration number as 10 and tune the initial candidate number to get the time-recall curve.

- 4) **kGraph**. kGraph [11] is an open library for ANN search based on  $k$ NN graph. The author of kGraph is the inventor of NN-descent [12]. The ANN search algorithm in kGraph is essentially the same as GNNS. The original Kgraph library implements with OpenMP (for parallelism) and SSE instructions for speed-up. We simply turn off the parallelism and SSE for fair comparison.
- 5) **IEH**. IEH is a short name for Iterative Expanding Hashing [21]. It is another ANN search method using  $k$ NN graph. Different from GNNS, IEH uses hashing methods to generate the initial result given a query. Considering the efficiency of hash coding, **IEH-LSH** and **IEH-ITQ** are compared in our experiment. The former uses LSH [16] as the hashing method and the latter uses ITQ [35] as the hashing method. Both hashing methods use 32 bit code. We also fix the iteration number as 10 and tune the initial result size to get the time-recall curve.
- 6) **Efanna**. The algorithm proposed in this paper. We use 16 trees for SIFT1M and 32 trees for GIST1M. The iteration number in NN-expansion stage is fixed as 4. We tune the search-to depth parameter  $S_{depth}$  and the candidate pool size  $P$  to get the time-recall curve.

Among the five compared ANN methods, Flann’s KD-tree is the hierarchical structure (tree) based method. The other four compared methods are all graph based methods. We do not compare with hashing based methods because [21] shows the significant improvement of IEH over the corresponding hashing methods.

All the graph based methods need a pre-built  $k$ NN graph and we use a ground truth 10-NN graph.

### 4.2.3 Results

The time-recall curves of all the algorithms on two data sets can be seen in Fig. 2 and Fig. 3. The index size of various algorithms are shown in Table 2. A number of interesting conclusions can be drawn as follows:

- 1) Our Efanna algorithm significantly outperforms all the other methods at all the cases on both of two data sets. Even at a relatively high recall (*e.g.*, 95%), Efanna is about 100x faster than the brute-force search on the SIFT1M and about 10x faster than the brute-force search on the GIST1M.
- 2) The GIST1M is a harder dataset than the SIFT1M for ANN search. At a relatively high recall (*e.g.*, 95%), all the ANN search methods are significantly faster than the brute-force search. However, on GIST1M some methods (flann, GNNS, kGrpah) are similar (or even slower) to the brute-force search. The reason may be the high dimensionality of the GIST1M.

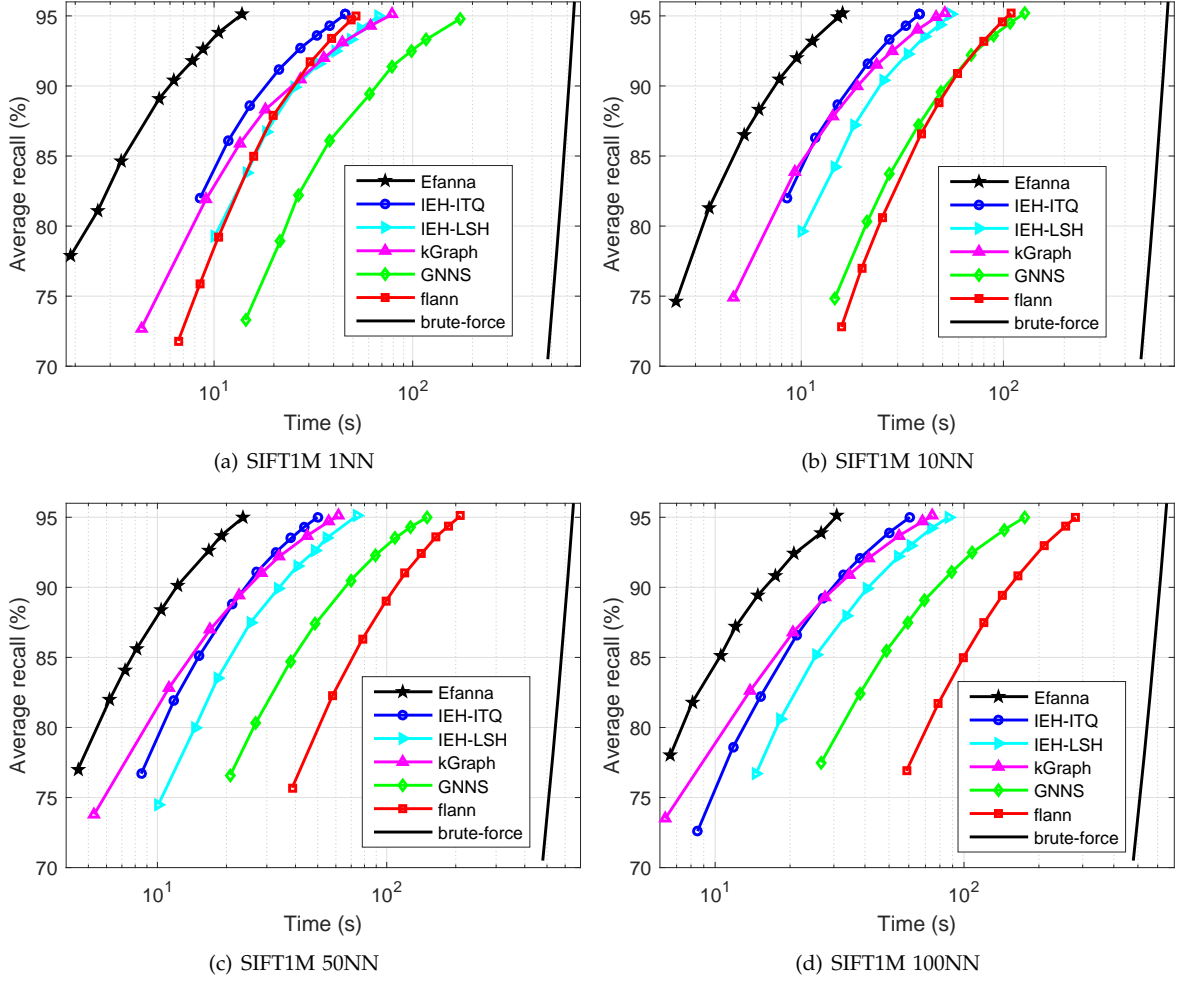


Fig. 2. ANN search results of 10,000 queries on SIFT1M. We use a 10-NN ground truth graph for all graph based methods.

TABLE 2  
Index Size of Different Algorithms

data set	algorithms	index size		
		tree (hash table)	graph	all
SIFT1M	flann(16-tree)	275 MB	0	275 MB
	Efanna(16-tree, 10-NN)	240 MB	40 MB	280 MB
	GNNS(10-NN)	0	40 MB	40 MB
	kGraph (10-NN)	0	40 MB	40 MB
	IEH-LSH (32 bit, 10-NN)	4 MB	40 MB	44 MB
	IEH-ITQ (32 bit, 10-NN)	4 MB	40 MB	44 MB
GIST1M	flann(32-tree)	550 MB	0	550 MB
	Efanna(32-tree, 10-NN)	494 MB	40 MB	534 MB
	GNNS(10-NN)	0	40 MB	40 MB
	kGraph (10-NN)	0	40 MB	40 MB
	IEH-LSH (32 bit,10-NN)	4 MB	40 MB	44 MB
	IEH-ITQ (32 bit,10-NN)	4 MB	40 MB	44 MB

3) When the required number of nearest neighbors is large (e.g., 50 and 100), all the graph based methods are significantly better than Flann's KD-tree. Since

50 or 100 results are more common in practical search scenarios, the graph based methods have the advantage.



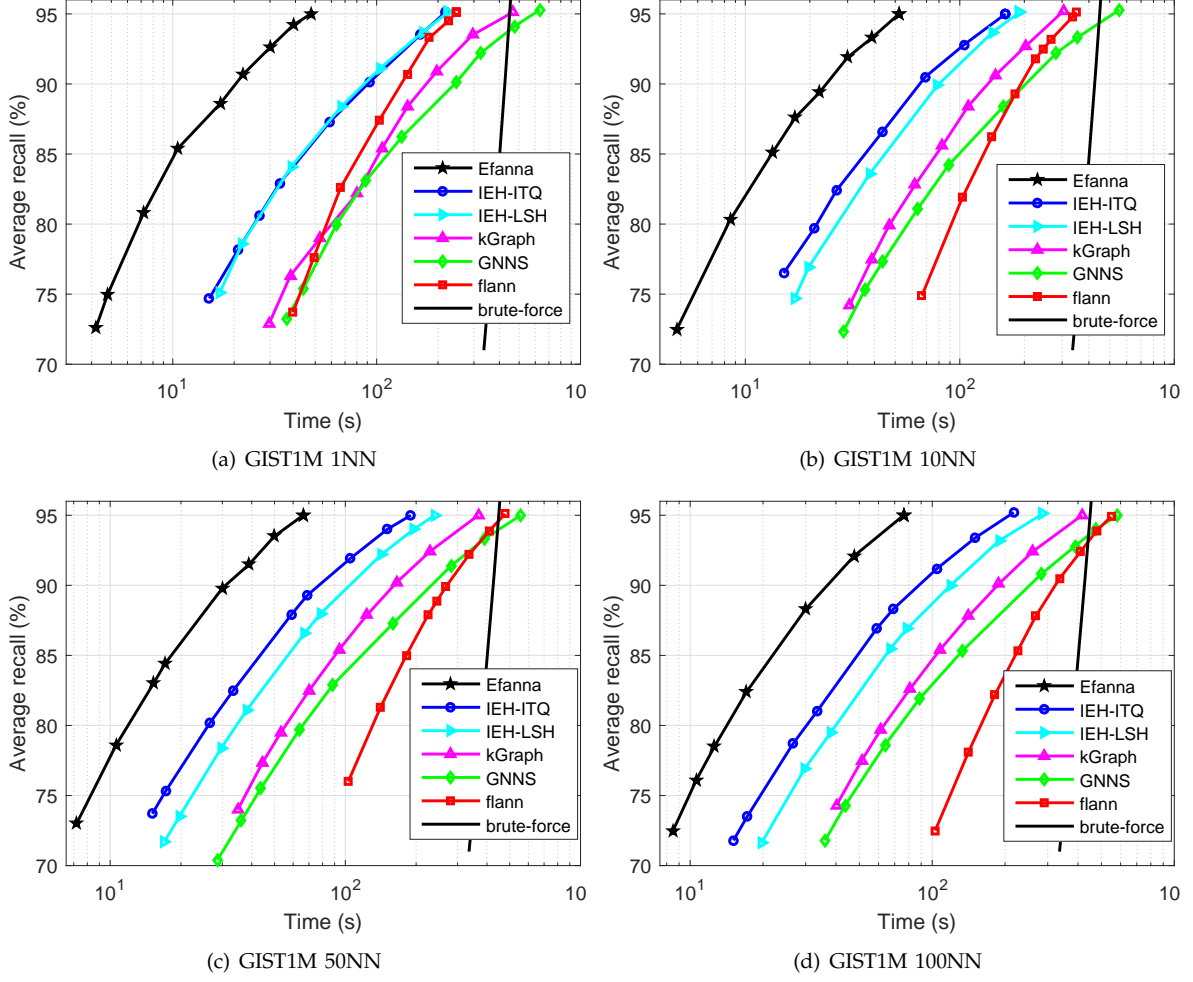


Fig. 3. ANN search results of 1,000 queries on GIST1M. We use a 10-NN ground truth graph for all graph based methods.

- 4) GNNS and kGraph are essentially the same algorithm. However, kGraph is significantly better than GNNS, particularly on SIFT1M. This is simply because the different data structure and implementations.
- 5) We implement four graph based methods (GNNS, IEH-LSH, IEH-ITQ and Efanna) exactly with the same framework. The only difference is the initialization: GNNS uses the random selection, IEH uses the hashing and Efanna uses the truncated KD-tree. The performance gap between these methods indicates the effectiveness of different initialization methods. The truncated KD-tree is better than the hashing and these two are better than the random selection. And the ITQ is better than the LSH.
- 6) TABLE 2 shows the index size of different algorithms. The index size of Efanna and that of Flann's KD-tree are comparable. Efanna's truncated KD-tree is slightly smaller than Flann's original KD-tree and Efanna needs an additional  $k$ NN graph. Because we use 16 trees in SIFT1M and 32 trees in GIST1M, the index size of Efanna and Flann are

significantly larger than other methods. To reduce the index size, one can use less trees in Efanna but maintain the high performance. We will discuss this in the section 4.5.

- 7) The index size of GNNS and KGraph is smallest because they only need to store a  $k$ NN graph. The index size of IEH is a little bit larger for storing the hashing code. For the scenarios that the index size is very crucial, IEH seems to be a promising method.
- 8) Considering both search performance and index size, graph based methods is a better choice than Flann's KD-tree.

### 4.3 Experiment on Approximate kNN Graph Construction

We show in last section that graph based methods can achieve very good performance on ANN search. However, the results above are based on a ground truth 10-NN graph. Table 3 shows the time cost to build the ground truth 10-NN graph for two datasets. It takes about 17 hours of CPU time on SIFT1M and about

TABLE 3  
Time of building the ground truth 10-NN graph on  
GIST1M and SIFT1M using brute-force search

data set	time (seconds)
SIFT1M	68,060
GIST1M	565,060

a week on GIST1M. Obviously, brute-force is not an acceptable choice. [17], [21] assume that the ground truth  $k$ NN graph exists. However, building the  $k$ NN graph is a step of indexing part of all the graph based methods. To make the graph based ANN search methods practically useful, we need to discuss how to build the  $k$ NN graph efficiently.

In this section, we will compare the performance of several approximate  $k$ NN graph construction methods.

#### 4.3.1 Evaluation Protocol

We use the accuracy-time curve to measure the performance of different approximate  $k$ NN graph construction algorithms. Given a data set with  $N$  points, an approximate  $k$ NN graph construction method should return  $N$  groups of  $k$  points, and each group of points stands for nearest neighbors the algorithm finds within the data set for the respective point. Suppose for point  $i$ , the returned point set of is  $R'_i$  and the ground truth set is  $R_i$ . Then the accuracy of point  $i$ ,  $accuracy_i$ , is defined as

$$accuracy_i = \frac{|R'_i \cap R_i|}{|R'_i|} \quad (2)$$

Then the *Accuracy* of the returned graph is defined as the average accuracy over all the  $N$  points:

$$Accuracy = \frac{\sum_i^N |R'_i \cap R_i|}{N|R'_i|} \quad (3)$$

We compare the performance of all the algorithms on building a 10-NN graph (*i.e.*, the sizes of  $R_i$  and  $R'_i$  are 10).

#### 4.3.2 Comparison Algorithms

- 1) **brute-force**: We report the performance of brute-force graph construction to show the advantages of using approximate  $k$ NN graph construction methods. To get different graph accuracy, we simply perform brute-force graph construction on different percentage of the data points.
- 2) **SGraph**: We refer to the algorithm proposed in [15] as SGraph. SGraph build the graph with three steps. First they generates initial graph by randomly dividing the data set into small ones iteratively and the dividing is carried out many times. Then they do brute-force graph construction within each subsets and combine all the subgraph into a whole. Finally they refine the graph using a technique similar to NN-expansion.

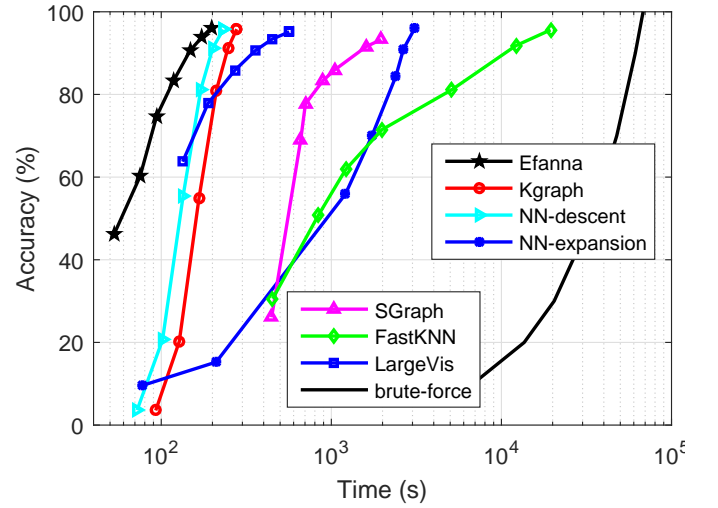


Fig. 4. 10-NN approximate graph construction results on SIFT1M

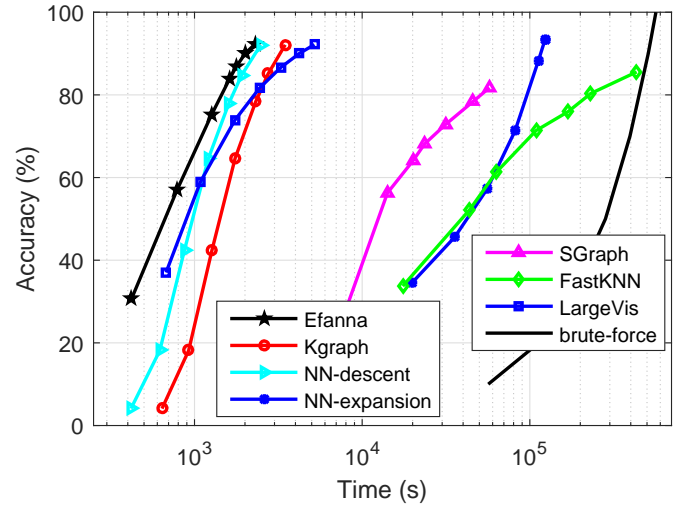


Fig. 5. 10-NN approximate graph construction results on GIST1M

- 3) **FastKNN**: We refer to the algorithm proposed in [36] as FastKNN. The last two steps of their graph building process is similar to SGraph. While FastKNN uses hashing method (specifically, AGH[22]) to generate the initial graph.
- 4) **NN-expansion**: The main idea of building approximate  $k$ NN graph with NN-expansion is to cast the graph construction problem as  $N$  ANN search problems, where  $N$  is the data size. However, NN-expansion is proposed for ANN search while not for AKNN graph construction. The reason we add it to the compared algorithms in this section is that some previous works [36] [30] claim to outperform NN-descent. While we find there may be misunderstanding that they may actually compared with NN-expansion rather than NN-descent.
- 5) **NN-descent** [12]: This algorithm first initializes the graph randomly. Then NN-descent refine it itera-

tively with techniques like local join and sampling [12]. Local join is to do brute-force searching within a point  $q$ 's neighbors which is irrelevant to  $q$ . Sampling is to ensure number of points involved in the local join is small, but the algorithm is still efficient.

- 6) **kGraph**: kGraph [11] is an open source library for approximate  $k$ NN graph construction and ANN search. The author of kGraph is the author of NN-descent. The approximate  $k$ NN graph algorithm implemented in kGraph library is exactly NN-descent. kGraph implements with OpenMP and SSE instruction for speed-up. For faire comparison, we disable the parallelism and SSE instruction.
- 7) **LargeVis** [30]: This algorithm is proposed for high dimension data visualization. The first step of LargeVis is to build an approximate  $k$ NN graph. LargeVis uses random projection tree and NN-expansion to build this graph.
- 8) **Efanna**: The algorithm proposed in this paper. We use hierarchical divide-and-conquer to get an initial graph. And then use NN-descent to refine the graph. In this experiments, we use 8 randomized truncated KD-trees to initialize the graph.

#### 4.3.3 Results

The time-accuracy curves of different algorithms on two data sets are shown in Fig. 4 and Fig. 5 receptively. A number of interesting conclusions can be made.

- 1) EFANNA outperforms all the other algorithms on approximate  $k$ NN graph building. It can achieve more than 300 times speed-up over brute-force construction to reach 95% accuracy. Without parallelism, it takes a week to build a 10-NN graph on GIST1M using brute-force search. Now the time can be reduced to less than an hour by using EFANNA.
- 2) We didn't get the source code of SGraph and FastKNN. So we implement their algorithms on our own. However, the performances shown in the two figures are quite different from what the original papers [15] [36] claim. One of the reasons may be the implementation. In the original FastKNN paper [36], the authors fail to add the hashing time into the total graph building time but actually should do. Fortunately, [15] reported that SGraph achieved 100 times speed-up over brute-force on the SIFT1M at 95% accuracy. And SGraph got 50 times speed-up over brute-force on the gist1M (384 dimensions) at 90% accuracy. While EFANNA achieves over 300 times speed-up on both SIFT1M and GIST1M (960 dimensions).
- 3) LargeVis achieve significant better result than NN-expansion. However, NN-descent is better than LargeVis, especially when we want an accurate graph. This results confirm our assumption that many previous works had the misunderstanding of NN-descent. The result reported in their paper

is actually NN-expansion [36], [30] rather than NN-descent.

- 4) kGraph and NN-descent are actually the same algorithm. The only difference is that we implement NN-descent by ourselves and kGraph is an open library. The performance difference of these two methods should due to the implementation.
- 5) The only difference between EFANNA and NN-descent (kGraph) is the initialization. EFANNA uses randomized truncated KD-tree to build the initial graph while NN-descent (kGraph) use random initialization.
- 6) The performance advantage of EFANNA over NN-descent is larger on the SIFT1M than on the GIST1M. The reason maybe the GIST1M (960 dimensions) has higher dimensionality than the SIFT1M (128 dimensions). The KD-tree initialization becomes less effective when dimensions becomes high. The similar phenomena happens when we compare EFANNA and LargeVis. Since LargeVis uses random projection trees for initialization, this suggests random projection trees maybe better than KD-tree when the dimensions is high. Using random projection trees as the hierarchical structures of EFANNA can be the future work.

#### 4.4 ANN Search with Approximate $k$ NN Graphs

The experimental results in the last section show that EFANNA can build an approximate  $k$ NN graph efficiently. However, there are no published results on the performance of graph based ANN search methods on an approximate  $k$ NN graph.

In this section, we evaluate the performance of EFANNA on approximate  $k$ NN graphs with various accuracy. The results on two data sets are shown in Fig .6 and 7 respectively.

From these two figures, we can see that the ANN search performance of EFANNA suffers from very little decrease in performance even when the graph is only "half right". Specifically, the ANN search preformance of EFANNA with a 60% accurate 10-NN graph is still significant better than Flann-kdtree on SIFT1M. On GIST1M, EFANNA with a 57% accurate 10-NN graph is significant better than Flann-kdtree.

These results are significant because building a less accurate  $k$ NN graph using EFANNA is very efficient. Table 4 shows the indexing time of EFANNA and Flann-kdtree. If a 60% accurate graph is used, the indexing time of EFANNA is less than that of Flann-kdtree. Combing the results in Table 2, we can see that comparing with Flann-kdtree, EFANNA takes less indexing time, similar index size and significant better ANN search performance.

Why EFANNA can get such a good ANN search performance even with a "half right" graph? Table 5 may explain the reason. The accuracy defined in Eqn. 2 uses the size of  $R$  and  $R'$ . The former is the true

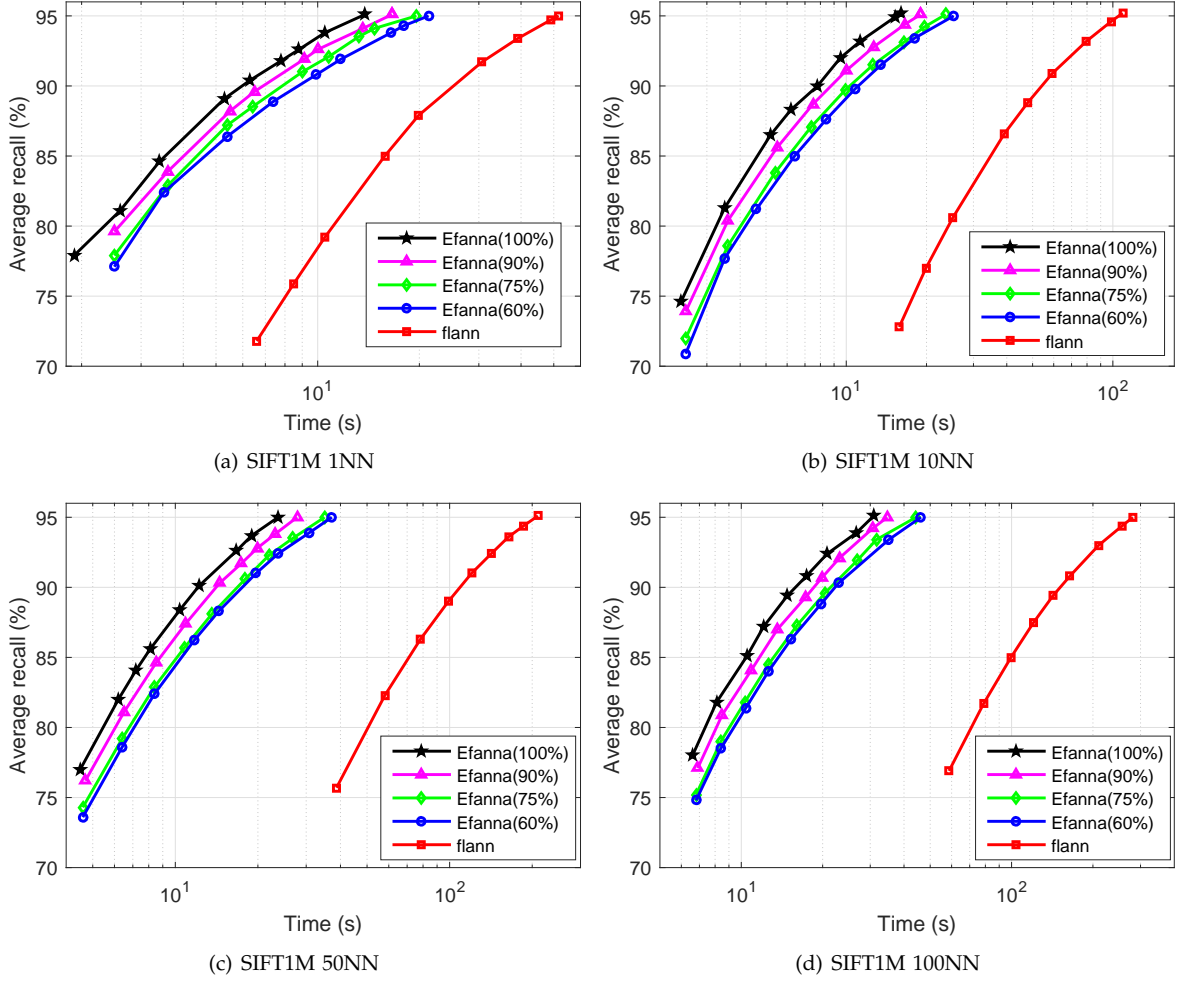


Fig. 6. Approximate nearest neighbor search results of 10,000 queries on SIFT1M. We use a 60% ~ 100% accuracy 10-NN graphs for EFANNA respectively.

TABLE 4  
Indexing Time of Efanna and flann

data set	algorithms	indexing time		
		tree building	graph building	all
SIFT1M	flann(16-tree)	131.7s	0	131.7s
	Efanna(16-tree, 90% 10-NN)	8.2s	148.0s	156.2s
	Efanna(16-tree, 75% 10-NN)	8.2s	93.9s	102.1s
	Efanna(16-tree, 60% 10-NN)	8.2s	75.1s	83.3s
GIST1M	flann(32-tree)	1423.5s	0	1423.5s
	Efanna(32-tree, 90% 10-NN)	67.0s	2017.7s	2084.7s
	Efanna(32-tree, 75% 10-NN)	67.0s	1267.7s	1334.7s
	Efanna(32-tree, 57% 10-NN)	67.0s	788.7s	855.7s

nearest neighbors set while the latter is the returned nearest neighbors set of an algorithm. In the previous experiments, we fix the sizes of both  $R$  and  $R'$  as 10. Table 5 reports the results by varying the size of  $R$  from 10 to 100. We can see that a 60% accurate 10-NN graph constructed by EFANNA in SIFT1M means 60% of all the neighbors are true 10-nearest neighbors. And

the remaining 40% neighbors are not randomly select from the whole dataset. Actually, 98.9% of the neighbors are true 100-nearest neighbors. These results show that the approximate  $k$ NN graphs constructed by EFANNA are very good approximation of the ground truth  $k$ NN graph.

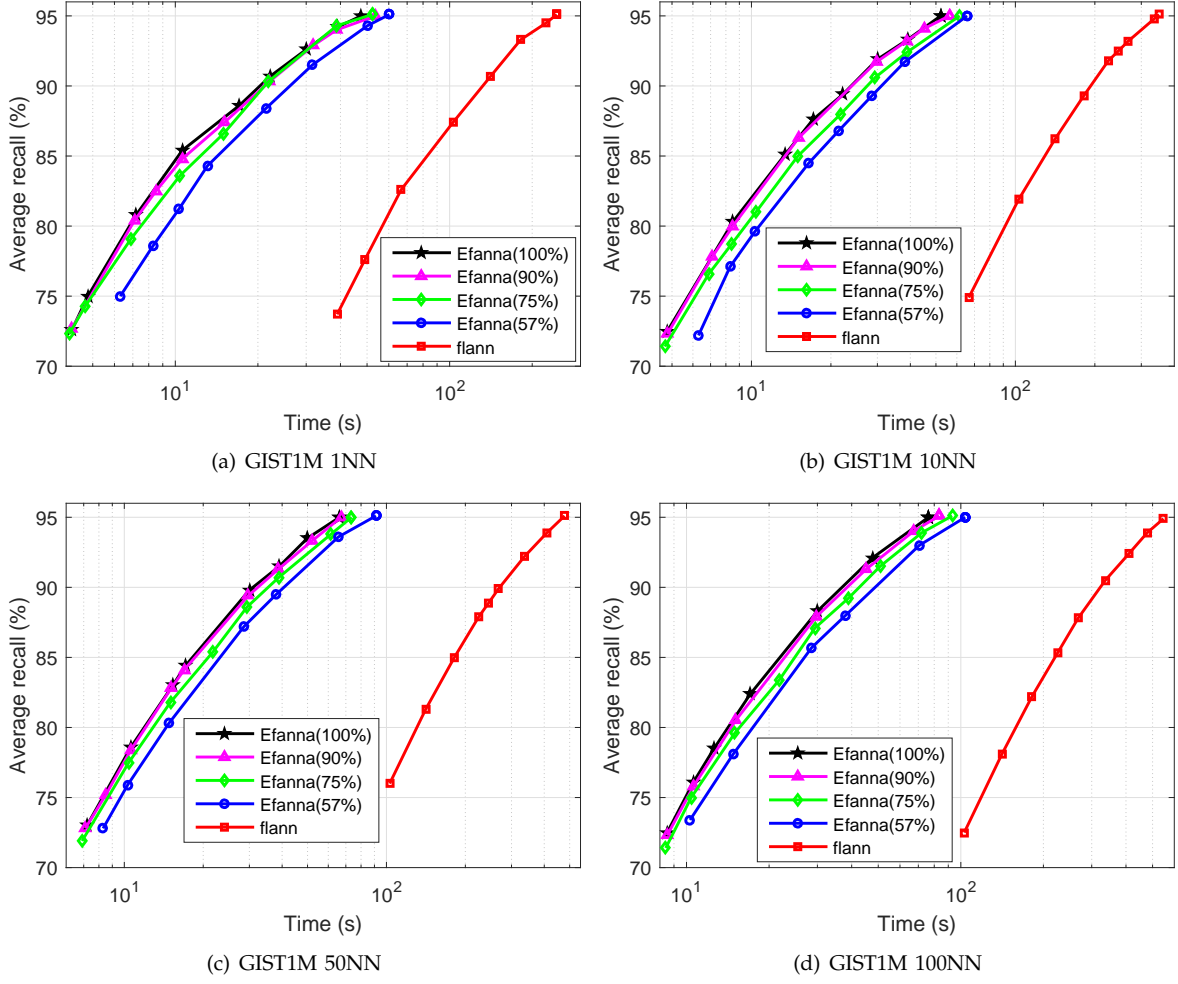


Fig. 7. Approximate nearest neighbor search results of 1,000 queries on GIST1M. We use a 57% ~ 100% accuracy 10-NN graphs for EFANNA respectively.

TABLE 5  
Efanna graph accuracy VS. k

data set	Efanna graph	Accuracy									
		10NN	20NN	30NN	40NN	50NN	60NN	70NN	80NN	90NN	100NN
SIFT1M	90% 10-NN	0.906515	0.99165	0.99788	0.999225	0.999654	0.999826	0.999901	0.999936	0.999955	0.999968
	75% 10-NN	0.747126	0.932613	0.971548	0.985388	0.991626	0.994829	0.996636	0.997721	0.998397	0.998838
	60% 10-NN	0.602044	0.823565	0.899977	0.936402	0.956686	0.969043	0.977085	0.982535	0.986419	0.989237
GIST1M	90% 10-NN	0.901412	0.994701	0.998917	0.999624	0.999828	0.999896	0.999921	0.999932	0.99994	0.999943
	75% 10-NN	0.75174	0.937751	0.974985	0.987682	0.993141	0.99586	0.99735	0.998223	0.998756	0.999094
	57% 10-NN	0.570855	0.792676	0.876249	0.91897	0.943883	0.959537	0.969966	0.97717	0.982319	0.986129

#### 4.5 ANN Search with Different Number of Trees

In the previous experiments, EFANNA uses 16 truncated kd-trees in SIFT1M and 32 trees in GIST1M for search initialization. Table 2 shows that these trees consume a large number of memory space. In this experiment, we want to explore how the number of trees will influence the performance of EFANNA on ANN search.

The ANN search results on SIFT1M and GIST1M are shown in Fig. 8 and Fig. 9 respectively. We simply

compare with IEH-ITQ and FLANN, because IEH-ITQ is the second best algorithm on ANN search in our previous experiment while FLANN also has the tree number parameter.

From Fig. 8 and 9, we can see that with less number of trees, the ANN search performances of both EFANNA and Flann decrease. However, with 4 trees on SIFT1M and 8 trees on GIST1M, EFANNA still significantly better than IEH-ITQ. While the index sizes can be significantly

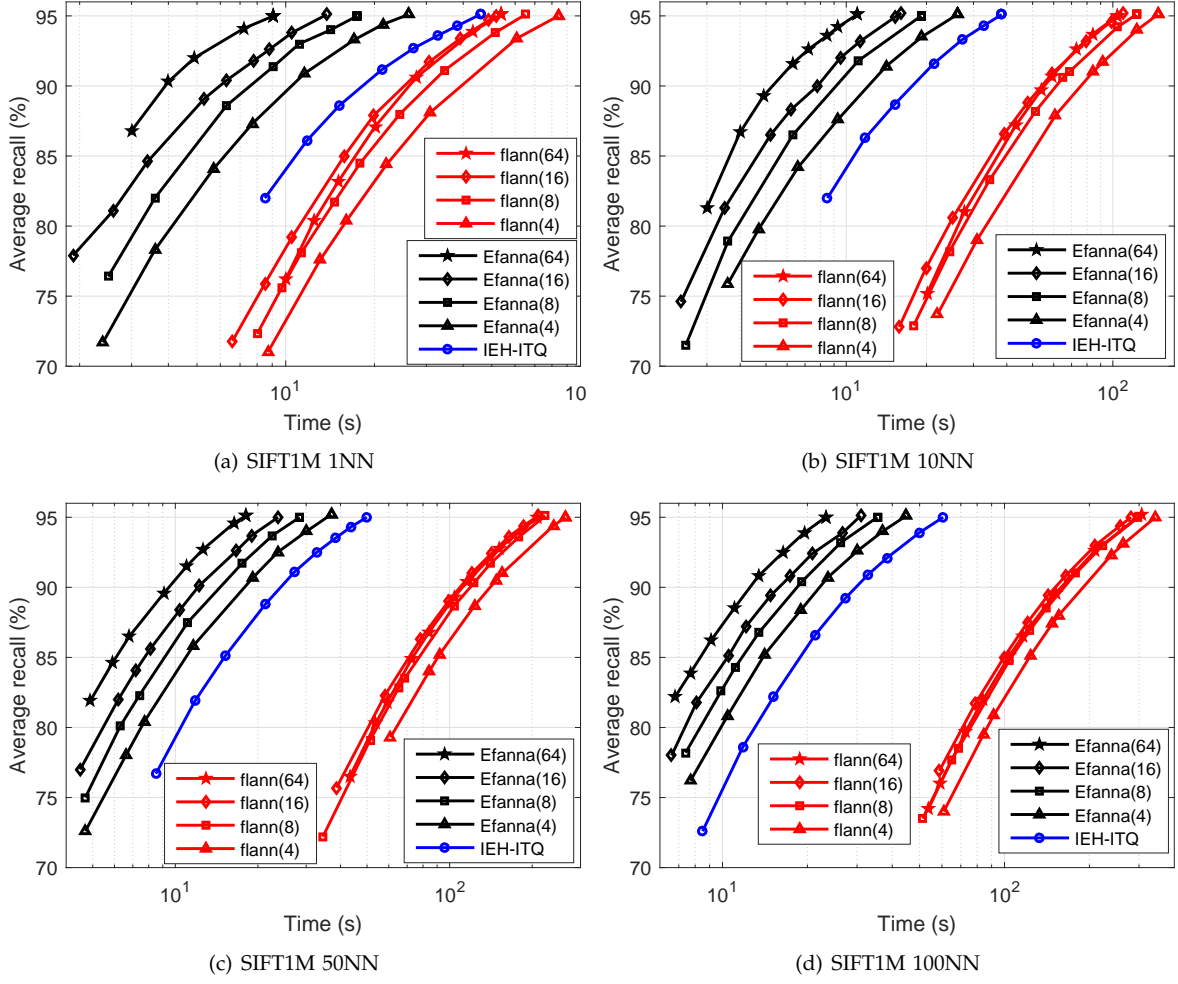


Fig. 8. Approximate nearest neighbor search results of 10,000 queries on SIFT1M. We use 4 ~ 64 trees for EFANNA and flann-kdtrees respectively.

TABLE 6  
Index Size of Efanna and Flann with Different Number of Trees

data set	algorithm	index size			algorithm	index size		
		tree	graph	all		tree	graph	all
SIFT1M	flann(64-tree)	1.1 GB	0	1.1 GB	Efanna(64-tree, 10-NN)	960 MB	40 MB	1 GB
	flann(16-tree)	275 MB	0	275 MB	Efanna(16-tree, 10-NN)	240 MB	40 MB	280 MB
	flann(8-tree)	138 MB	0	138 MB	Efanna(8-tree, 10-NN)	120 MB	40 MB	160 MB
	flann(4-tree)	69 MB	0	69 MB	Efanna(4-tree, 10-NN)	61 MB	40 MB	101 MB
GIST1M	flann(64-tree)	1.1 GB	0	1.1 GB	Efanna(64-tree, 10-NN)	987 MB	40 MB	1.027 GB
	flann(32-tree)	550 MB	0	550 MB	Efanna(32-tree, 10-NN)	494 MB	40 MB	534 MB
	flann(16-tree)	275 MB	0	275 MB	Efanna(16-tree, 10-NN)	247 MB	40 MB	287 MB
	flann(8-tree)	138 MB	0	138 MB	Efanna(8-tree, 10-NN)	124 MB	40 MB	164 MB

reduced as suggested by Table 6. We can also see that when increasing the number of trees, EFANNA's performance improves much more than FLANN does.

The results in this section show the flexibility of EFANNA over other graph based ANN search methods. One can easily make trade-off between index size and search performance.

## 5 THE EFANNA LIBRARY

The work in this paper is released as an open source library. Please access the code at Github<sup>2</sup>. The openMP supported version will be released soon.

2. <https://github.com/fc731097343/efanna>



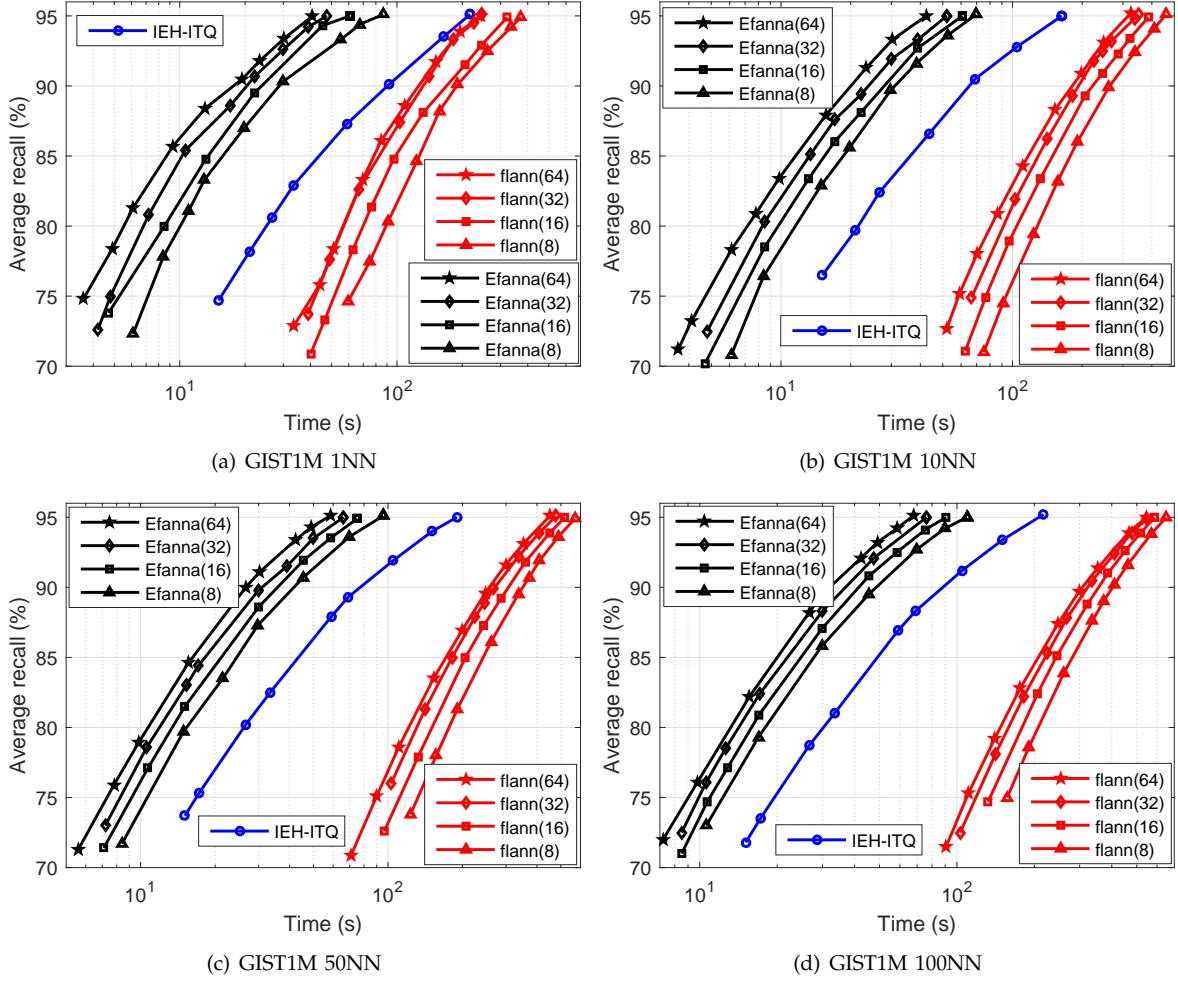


Fig. 9. Approximate nearest neighbor search results of 1,000 queries on GIST1M. We use 8 ~ 64 trees for EFANNA and flann-kdtrees respectively.

## 6 CONCLUSION

The goal of this research is to provide a fast solution, EFANNA, for ANN search problem and approximate  $k$ NN graph construction. On ANN search, we use hierarchical structures to provide better initialization for NN-expansion. And on graph construction, we use a divide-and-conquer way to construct an initial graph and refine it with NN-descent. Extensive experiments shows that EFANNA outperforms previous algorithms significantly both in approximate  $k$ NN graph construction and ANN search. Meanwhile, EFANNA also shows great flexibility for different scenarios.

## REFERENCES

- [1] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Y. Wu. An optimal algorithm for approximate nearest neighbor searching fixed dimensions. *Journal of the ACM (JACM)*, 45(6):891–923, 1998.
- [2] M. Belkin, P. Niyogi, and V. Sindhwani. Manifold regularization: A geometric framework for learning from labeled and unlabeled examples. *Journal of Machine Learning Research*, 7(Nov):2399–2434, 2006.
- [3] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the Acm*, 18(9):509–517, 1975.
- [4] J. L. Bentley. Multidimensional divide-and-conquer. *Communications of the Acm*, 23(4):214–229, 1980.
- [5] N. Bhatia and Vandana. Survey of nearest neighbor techniques. *arXiv preprint arXiv:1007.0085*, 2010.
- [6] E. Bingham and H. Mannila. Random projection in dimensionality reduction: applications to image and text data. In *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 245–250, 2001.
- [7] S. Brin. Near neighbor search in large metric spaces. In *Proceedings of the 21th International Conference on Very Large Data Bases*, pages 574–584, 1995.
- [8] J. Chen, H. R. Fang, and Y. Saad. Fast approximate knn graph construction for high dimensional data via recursive lanczos bisection. *Journal of Machine Learning Research*, 10(Sep):1989–2012, 2009.
- [9] K. L. Clarkson. Fast algorithms for the all nearest neighbors problem. *Foundations of Computer Science Annual Symposium on*, pages 226–232, 1983.
- [10] S. Dasgupta and Y. Freund. Random projection trees and low dimensional manifolds. In *Proceedings of the fortieth annual ACM symposium on Theory of computing*, pages 537–546. ACM, 2008.
- [11] W. Dong. Kgraph, an open source library for k-nn graph construction and nearest neighbor search. [www.kgraph.org](http://www.kgraph.org), 2014.
- [12] W. Dong, C. Moses, and K. Li. Efficient k-nearest neighbor graph construction for generic similarity measures. In *Proceedings of the 20th international Conference on World Wide Web*, pages 577–586, 2011.

- [13] J. H. Friedman, J. L. Bentley, and R. A. Finkel. An algorithm for finding best matches in logarithmic expected time. *Acm Transactions on Mathematical Software*, 3(3):209–226, 1975.
- [14] K. Fukunaga and P. M. Narendra. A branch and bound algorithm for computing k-nearest neighbors. *IEEE Transactions on Computers*, 100(7):750–753, 1975.
- [15] R. Gan, J. Wang, J. Wang, G. Zeng, Z. Tu, and S. Li. Scalable k-nn graph construction for visual descriptors. In *Proceedings of the 2012 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1106–1113, 2012.
- [16] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *Proceedings of the 25th International Conference on Very Large Data Bases*, pages 518–529, 1999.
- [17] K. Hajebi, Y. Abbasi-Yadkori, H. Shahbazi, and H. Zhang. Fast approximate nearest-neighbor search with k-nearest neighbor graph. In *IJCAI 2011, Proceedings of the International Joint Conference on Artificial Intelligence*, volume 22, pages 1312–1317, 2011.
- [18] J. P. Heo, Y. Lee, J. He, S. F. Chang, and S. E. Yoon. Spherical hashing. In *Proceedings of the 2012 IEEE Conference on Computer Vision and Pattern Recognition*, pages 2957–2964, 2012.
- [19] P. Indyk and R. Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pages 604–613, 1998.
- [20] Z. Jin, Y. Hu, Y. Lin, D. Zhang, S. Lin, D. Cai, and X. Li. Complementary projection hashing. In *Proceedings of the 2013 IEEE International Conference on Computer Vision (ICCV)*, pages 257–264, 2013.
- [21] Z. Jin, D. Zhang, Y. Hu, S. Lin, D. Cai, and X. He. Fast and accurate hashing via iterative nearest neighbors expansion. *IEEE transactions on cybernetics*, 44(11):2167–2177, 2014.
- [22] W. Liu, J. Wang, S. Kumar, and S.-F. Chang. Hashing with graphs. In *Proceedings of the 28th international conference on machine learning (ICML-11)*, pages 1–8, 2011.
- [23] J. Makhoul, F. Kubala, R. Schwartz, and R. Weischedel. Performance measures for information extraction. In *Proceedings of Darpa Broadcast News Workshop*, pages 249–252, 2000.
- [24] M. Muja and D. G. Lowe. Fast approximate nearest neighbors with automatic algorithm configuration. In *Visapp International Conference on Computer Vision Theory and Applications*, pages 331–340, 2009.
- [25] M. Muja and D. G. Lowe. Scalable nearest neighbor algorithms for high dimensional data. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 36(11):2227–2240, 2014.
- [26] D. Nister and H. Stewenius. Scalable recognition with a vocabulary tree. In *Proceedings of the 2006 IEEE Conference on Computer Vision and Pattern Recognition*, volume 2, pages 2161–2168, 2006.
- [27] P. C. of k-Nearest Neighbor Graphs in Metric Spaces. Paredes, rodrigo and Chávez, edgar and figueroa, karina and navarro, gonzalo. In *International Workshop on Experimental and Efficient Algorithms*, pages 85–97, 2006.
- [28] J. Shi and J. Malik. Normalized cuts and image segmentation. *IEEE Transactions on pattern analysis and machine intelligence*, 22(8):888–905, 2000.
- [29] C. Silpa-Anan and R. Hartley. Optimised kd-trees for fast image descriptor matching. In *Proceedings of the 2008 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–8, 2008.
- [30] J. Tang, J. Liu, M. Zhang, and Q. Mei. Visualizing large-scale and high-dimensional data. In *Proceedings of the 25th International Conference on World Wide Web*, pages 287–297, 2016.
- [31] P. M. Vaidya. An  $o(n \log n)$  algorithm for the all-nearest-neighbors problem. *Discrete & Computational Geometry*, 4(2):101–115, 1989.
- [32] J. Wang, H. T. Shen, J. Song, and J. Ji. Hashing for similarity search: A survey. *arXiv preprint arXiv:1408.2927*, 2014.
- [33] Y. Weiss, A. Torralba, and R. Fergus. Spectral hashing. In *Advances in neural information processing systems*, pages 1753–1760, 2009.
- [34] P. N. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms*, volume 93, pages 311–321, 1993.
- [35] G. Yunchao, L. Svetlana, G. Albert, and P. Florent. Iterative quantization: a procrustean approach to learning binary codes for large-scale image retrieval. In *Proceedings of the 2011 IEEE Conference on Computer Vision and Pattern Recognition*, pages 2916–2929, 2011.
- [36] Y.-m. Zhang, K. Huang, G. Geng, and C.-l. Liu. Fast knn graph construction with locality sensitive hashing. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 660–674. Springer, 2013.